

Working with Linux

One of the biggest stumbling blocks when writing software for Linux is understanding what Linux is and is not. Linux means different things to different people. Technically, Linux itself is an operating system kernel written by the Finnish born Linus Torvalds, though most people today casually refer to an entire Linux-based system by the same name. In just a few years, Linux has risen from obscurity and become widely accepted by some of the largest and most powerful computing users on the planet.

Linux is now a big-money, enterprise-quality operating system. It's used in some of the largest supercomputers and also many of the smallest gadgets, which you would never expect to have Linux underneath. Yet for all its prevalence — for such a big name in modern computing — Linux isn't owned by any one corporation that pulls the strings. Linux is so successful because of the many thousands of developers around the world who constantly strive to make it better. They, like you, are interested in writing high-quality software that draws upon the experience of others within the Linux community.

Whatever Linux means to you, you're reading this book because you're interested in learning more about becoming a professional Linux programmer. As you embark on this journey, you will find it helpful to tool yourself up with an understanding of the different flavors of Linux, how to get going in developing for them, and how working with Linux differs from working with many other popular platforms on the market today. If you're already a Linux expert, you need only skim this chapter. If you're working toward becoming the next expert, this chapter should provide some useful pointers.

In this chapter, you will learn what Linux is and how the individual components of a Linux distribution fit together, from a professional programmer's point of view. You will learn about the development process behind much of the Free, Libre, and Open Source Software (FLOSS) that is used on Linux systems and discover the wealth of online communities that power the open source revolution. Finally, you'll also discover a few of the ways in which Linux differs from other operating systems you've encountered in the past — more on that throughout the rest of the book, too.

A Brief History of Linux

Linux has a very diverse and interesting history, which dates back much further than you may at first think. In fact, Linux has heritage spanning more than 30 years, since the earliest UNIX systems of the 1970s. This fact isn't just relevant to die-hard enthusiasts. It's important for you to have at least a general understanding of the unique history that has led to the modern Linux systems that you will encounter today. Doing so will better equip you to understand the little idiosyncrasies that differentiate Linux from alternatives on the market — and help to make Linux development more interesting, too.

The earliest work on Linux itself began back in the summer of 1991, but long before there was Linux, there was the GNU project. That project had already spent well over a decade working on producing much of the necessary Free Software components in order to be able to create a fully Free operating system, such as Linux. Without the GNU project, Linux could never have happened — and without Linux, you might not be reading about the GNU project right now. Both projects have benefited enormously from one another, as you'll discover in the topics throughout this book.

The GNU Project

Back in 1983, Richard Stallman (aka RMS) was working in the artificial intelligence (AI) lab at MIT. Up until that time, many software applications had been supplied in source code form, or otherwise had source code available that users could modify for their own systems, if it was necessary. But at this time, it was a growing trend for software vendors to ship only binary versions of their software applications. Software source code had quickly become the “trade secret” of corporations, who would later become highly protective of their — what open source developers now often term — “secret sauce.”

The initial goal of the GNU project was to produce a Free UNIX-like operating system, complete with all of the necessary tools and utilities necessary in order to build such a system from source. It took well over a decade to produce most of the tools and utilities needed, including the GCC compiler, the GNU emacs text editor, and dozens of other utilities and documentation. Many of these tools have become renowned for their high quality and rich features — for example, GCC and the GNU debugger.

GNU enjoyed many early successes, but it had one crucial missing component throughout the 1980s. It had no kernel of its own — the core of the operating system — and instead relied upon users installing the GNU tools within existing commercial operating systems, such as proprietary UNIX. Though this didn't bother many of the people who used the GNU tools and utilities on their proprietary systems, the project as a whole could never be complete without a kernel of its own. There was intensive debate for years over alternatives (such as the developmental GNU HURD), before Linux came along.

Linux has never truly formed part of the GNU operating system that Richard Stallman had envisioned. In fact, for many years the GNU project has continued to advocate the GNU HURD microkernel over the Linux kernel in its conceptual GNU system, despite the fact that Linux has become the poster child for a new generation of users and developers and is by far more popular. Nevertheless, you will still occasionally see the term “GNU/Linux” used to refer to a complete Linux system in recognition of the large part played by the many GNU tools in both building and running any modern Linux system.

The Linux Kernel

The Linux kernel came along much later than the GNU project itself, over a decade after Richard Stallman made his initial announcement. In that time, other alternate systems had been developed. These included the HURD microkernel (which has since garnered limited general interest outside of the enthusiastic core developer community), as well as the educational Minix microkernel that had been written by Andrew Tanenbaum. For various reasons, neither of these alternative systems was widely considered ready for prime time by general computer users when Linux first hit the scene.

Meanwhile, a young Finnish student, working at the University of Helsinki had become frustrated about many of the things that he considered broken in the Minix operating system.¹ Thus, he began work on his own system, designed specifically for his (at the time cutting-edge) AT-386 microcomputer. That person was Linus Torvalds, and he would go on to lead the project that has created a whole industry of Linux companies and spurred on a new generation.

Linus sent out the following announcement to the comp.os.minix Usenet newsgroup upon the initial release of Linux in the summer of 1991:

Date: 25Aug 91 20:57:08 GMT

Organization: University of Helsinki

Hello everybody out there using minix – I’m doing a (free) Operating system (just a hobby, won’t be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I’d like any feedback on Things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I’ve currently ported bash (1.08) and gcc(1.40), and things seem to work. This implies that I’ll get something practical within a few months, and I’d like to know what features most people would want. Any Suggestions are welcome, but I won’t promise I’ll implement them.

Despite Linus’s initial modesty, interest in the Linux kernel grew quickly around the world. It wasn’t long before several release cycles had passed and a growing community of users — all of whom were necessarily developers; simply installing Linux required a great deal of expertise — were working to solve technical challenges and implement new ideas as they were first conceived. Many of the now infamous Linux developers became involved early on. They enjoyed being able to work on a modern entirely Free UNIX-like system that didn’t suffer from design complexities of alternative systems.

Linux developers relied upon the many existing GNU tools to build the Linux kernel and to develop new features for it. Indeed, it wasn’t long before interest grew beyond the early developers, and Minix users began to work on Linux instead — something that ultimately led to a series of well-known “flame wars” between the creator of Minix (Andrew Tanenbaum) and Linus Torvalds. Tanenbaum maintains to this day that the design of Linux is fundamentally inferior to that of Minix. Philosophically, this may be true, but the same can be said of other modern operating systems.

You can learn more about the historical heritage of Linux and other UNIX-like operating systems in the book *A Quarter Century of UNIX* by Peter H. Salus (Addison-Wesley, 1994).

¹Many of these issues remained for a number of years and would prove the topic of a large amount of conversation on the early Minix and Linux newsgroups. In latter years, the rivalry has largely subsided as Linux has asserted its dominance in the marketplace and Minix (and its various successors) has continued to be of academic interest to those contemplating future Operating System design.

Linux Distributions

With the growing popularity of the Linux kernel came an interest in making Linux more accessible to those who didn't already happen to have advanced knowledge of its internal programming. To create a usable Linux system, you need more than just the Linux kernel alone. In fact, the average Linux desktop system available today makes use of many thousands of individual software programs in order to go from system power on to a feature-rich graphical desktop environment such as GNOME.

When Linux was first released, there wasn't such a rich multitude of software available. In fact, Linus started out with just one application—the GNU Borne Again SHell (bash). Those who have ever had to boot a Linux or UNIX system into a limited “single-user” mode (where only a bash shell is run) will know what this experience feels like. Linus did much of his early testing of Linux from within a solitary bash command shell, but even that didn't just magically run on Linux; it first had to be *ported*, or modified to run on a Linux system instead of an existing system, such as Minix.

As more and more people began to use and develop software for Linux, a wide range of software became available to those with the patience to build and install it. Over time, it became apparent that building every single Linux system from scratch was an unsupportable, nonupgradeable nightmare that prevented all but the most enthusiastic from experiencing what Linux had to offer. The solution came in the form of Linux distributions, or precreated collections of applications and a Linux kernel that could be supplied on floppy disk (and later on CD) to a wide range of potential users.

Early Linux distributions were simply a convenience for those who wanted to avoid building the entire system from scratch for themselves, and did little to track what software had been installed or handle the safe removal and addition of new software. It wasn't until package managers like Red Hat's RPM and Debian's dpkg had been invented that it was possible for regular users to install a Linux system from scratch without very detailed expert knowledge. You'll discover more about package management in later in the book, when you look at building your own prepackaged Linux software for distribution.

Modern Linux distributions come in many shapes and sizes and are targeted at a variety of different markets. There are those written for regular desktop Linux users; those written for enterprise users with demands of scalable, robust performance; and even distributions designed for embedded devices such as PDAs, cellular telephones and set-top boxes. Despite the different packaging, Linux distributions usually have commonalities that you can exploit. For example, most distributions strive to be compatible on some level with the Linux Standard Base (LSB) *de facto* set of standards for compatible Linux environments.

Free Software vs. Open Source

Richard Stallman started the GNU project and founded the Free Software Foundation as a nonprofit organization to oversee it. He also worked on the first version of the General Public License—the GPL—under which a large proportion of software written for systems that run Linux is licensed. The GPL is an interesting document in its own right because its goal is not to restrict your use of GPL licensed software, but to protect the right of users and developers to have access to the source code.²

²The GPL is currently undergoing its third major rewrite at the time that this book is being written. The new version is likely to be one of the most controversial Free Software licenses yet. It includes stipulations about the licensing of patents and other technology, attempts to outlaw Digital Rights Management (termed “Digital Restrictions Management” by Richard Stallman) and a great deal of other requirements besides.

The GPL allows you to make changes to the Linux kernel and other GPL-licensed Free Software, in return for you publishing those changes so that other people may use them (or incorporate them back into the next official release of a given piece of software). For example, the GPL allows you to fix a bug in a major application such as Open Office, or to add custom audio file support to the totem multimedia player on a GNOME desktop system. The GPL affords you, as a developer, a great deal of flexibility to use Linux for whatever purpose you choose, just as long as you make your modifications available for others to do likewise. That's the key point — the GPL tries to keep the development process open.

Unfortunately for Richard Stallman, the English language isn't well equipped with an equivalent of the French word *libre* (free as in liberty), so many people confuse the concept of Free Software with software that is monetarily free. In fact, much Free Software is entirely free of charge, but there are also companies who make money from selling GPL-licensed software (including its freely redistributable source code). They are able to make money not through the software itself, but by offering a variety of support options and additional professional services for when things go wrong.

To reduce the confusion associated with the term “Free Software,” the term “Open Source” was coined and became popular during the 1990s. Unlike Free Software, open source does not specifically refer to GPL-licensed software. Instead, it refers to the general desire for software to come with source code included (so that it can be tuned, debugged, and improved by others), even if that source code is actually under a more restrictive license than the GPL itself. For this reason, there is a lot more software available technically meeting the definition of open source, while simultaneously not being Free.

It is very important that you have an understanding of the requirements that the GPL places on the work that you may do in modifying existing GPL-licensed software. Although you are not required to use the GPL in your own programs, you must respect the rights of others who have done so. There are numerous examples of potential GPL infringement on the Internet — usually from companies who didn't know that they needed to make their modifications to software such as the Linux kernel available for others to take a look at. You don't want to become the next example case, so always ensure that both you and your colleagues are aware of the GPL, and decide early on how you want to work with it.

Beginning Development

The first step you take as a Linux developer is to tool yourself up for the tasks that lie ahead. This means that you'll need to have a suitable development system available to you on which you can compile and test your own Linux programs. Almost any reasonable workstation will be sufficient, at least at first — though if you end up building a lot of software, you might elect for a higher-performance machine to reduce build times. There's little else more demotivating than constantly waiting for large software builds to complete. Still, it's always good to walk before you try to run.

It should be stressed at this point that the authors of this book are not going to suggest to you that you install or use a particular Linux distribution. There are plenty of good alternatives out there, and it's the job of corporate marketing and community interest to convince you of the merits of working with and supporting a particular set of Linux distributions over any others. Nonetheless, it does make sense to look at well-known distributions (at least at first) so that you'll have better access to a highly active community of developers who can help you as and when you make your first mistakes.

You can track the current trends in modern Linux distributions through impartial websites, such as www.distrowatch.com. Distrowatch also provide useful informational resources about each one.

Choosing a Linux Distribution

At the time that this book is being written, there are well over 300 Linux distributions in use around the world, and that number is growing almost daily. Since most (if not all) of the software shipped in the average Linux distribution is covered by the GNU General Public License (GPL), literally anyone can take that software and package it for themselves into their own distribution. This encourages initiative and experimentation, but it would also quickly lead to an unmanageable support nightmare for those who decided to package software for use by those with the 300 different distributions in use.

Fortunately for you as a software developer, most of the Linux users you will need to support are using a mere handful of popular Linux distributions. Those who are not apparently using one of these well-known distributions may well have a distribution that is based upon one. It's very common for newer distributions to be built upon the niche requirements of a subset of existing users. Obviously, it stands to reason that the 100 people using a particular specialist Linux distribution may not necessarily receive the same level of support as the many hundreds of thousands of people who use another.

Here are 10 of the more popular Linux distributions available today:

- ☐ Debian GNU/Linux
- ☐ Fedora (previously known as Fedora Core)
- ☐ Gentoo Linux
- ☐ Mandriva Linux
- ☐ Red Hat Enterprise Linux (RHEL)
- ☐ Slackware Linux
- ☐ OpenSuSE
- ☐ SuSE Linux Enterprise Server (SLES)
- ☐ Ubuntu

Linux Distributions from Red Hat

Red Hat once produced a version of Linux known as Red Hat Linux (RHL). This was available up until release 9.0, at which point the commercial product became known as Red Hat Enterprise Linux. Around the same time, the Fedora community Linux distribution became available for those who would prefer an entirely open source version without commercial support. Fedora is very popular with desktop users and enthusiasts and is widely used by Free Software developers, as well as commercial vendors — who will later need to test and certify their software against the Enterprise release as a separate endeavor.

For more information about Red Hat, see www.redhat.com. The Fedora project has a separate website, www.fedoraproject.org.

Linux Distributions from Novell

Novell bought SuSE in 2004 and gained full control over SuSE Linux. At around the same time, a variety of marketing and branding decisions affected the future naming of Linux products from Novell. Like Red Hat, Novell provide a community release of their operating system — known as OpenSUSE. It is maintained by a growing community of users, who help to cultivate new technologies that may

ultimately feed back into the next release of the commercial SuSE Linux Enterprise Server. Red Hat and Novell are usually considered to be the two big commercial Linux vendors in the marketplace.

For more information about Novell and SuSE, see www.novell.com. The OpenSUSE project has a separate website, www.opensuse.org.

Debian and Ubuntu GNU/Linux

Debian has been around for as long as Red Hat and SuSE and has a large group of core supporters. As an entirely community-maintained distribution, it is not motivated by the goals of any one particular corporation but strives simply to advance the state of the art. This is a laudable goal indeed, though Debian has suffered in the past from extremely large development cycles — often many years between major releases. A variety of “Debian derivatives” have been produced in the past, including Progeny Linux, which was one of the first attempts at producing a commercial version of Debian.

Mark Shuttleworth, one-time founder of Thwate made a fortune developing a business that had some reliance on Debian systems. Thus, he was heavily involved in the Debian community, and in 2004 founded the Ubuntu project. Ubuntu is based upon Debian, but it doesn’t aim to replace it. Rather, the goal of the Ubuntu project is to provide stable release cycles and productize Debian into a distribution for the masses. Canonical, the company backing Ubuntu development has developed various tools as part of this process, including the Launchpad and Rosetta tools mentioned later in this book.

For more information about Debian GNU/Linux, see www.debian.org. The Ubuntu project has a separate website, www.ubuntulinux.org.

Classes of Linux Distribution

Distributions can be broadly broken down into three different classes, depending upon their goals, whether they are a derivative of another popular distribution, and whether they are designed for ease of use or for those with more advanced requirements. For example, the average desktop user is unlikely to rebuild his or her entire Linux distribution on a whim, whereas some server administrators actively enjoy the power and flexibility of squeezing every last possible drop of performance out of their machines.

It’s important to remember that Linux delivers great flexibility — if someone can think of a way to use Linux and create a new distribution, somebody else is probably already working on implementing it.

RPM based Distributions

RPM-based distributions are so called because they use Red Hat’s RPM package management tools in order to package and distribute the individual components of the distribution. In early fall 1995, RPM was one of the first package management tools available for Linux. It was quickly adopted by other distributions, such as SuSE Linux. RPM has since been renamed from Red Hat Package Manager to RPM Package Manager — reflecting the independent development of the RPM tools happening today — but a number of distributions using RPM continue to share commonalities with Red Hat distributions.

RPM-based distributions such as Red Hat’s Enterprise Linux (RHEL) and Novell’s SuSE Linux Enterprise Server (SLES) make up a bulk of commercial Linux offerings used throughout the world today. If you’re writing software for use in the enterprise, you’ll want to ensure that you have support for RPM-based distributions such as these. You needn’t buy a copy of the Enterprise version of these distributions simply for everyday software development. Instead, you can use one of the community-maintained releases of the Fedora (Red Hat Linux derived) or OpenSUSE Linux distributions.

Chapter 1: Working with Linux

Debian Derivatives

As you will discover later in this book, Debian-derived distributions are based on the Debian Linux distribution and package management tools such as apt. Debian's dpkg package management tool was written around the same time that the original work was done on RPM, although different design decisions and philosophy have seen the two tools continue along separate paths ever since. Debian has a reputation for forming the basis of a variety of community and commercial Linux distributions.

Debian is a community-maintained Linux distribution, coordinated by a nonprofit organization known as Software in the Public Interest (SPI). Since the earliest releases, there has been an interest in customizing Debian and in distributing variants aimed at addressing a particular need. One of the most high-profile Debian derivatives is the Ubuntu Linux distribution, which aims to encourage widespread adoption through regulated release cycles and by steering overall development to meet certain goals.

Source Distributions

Linux distributions don't need to be based upon one of the common package management systems. There are many alternatives out there that use little or no package management beyond keeping software components in separate file archives. In addition, there are distributions that are actually intended for you to build when they are installed. This can be the case for any number of practical (or ideological) reasons but such distributions are usually confined to very niche Linux markets.

Build-from-source distributions such as Gentoo are designed to be easy to use but at the same time deliver high performance through locally customized software for each installed system. Gentoo uses a system known as portage to automate the process of downloading and building each individual software application whenever you require. Just bear in mind that it can take many hours for Open Office to build the first time you decide you need to use it and instruct portage to build it up for you.

You won't usually concern yourself with source-based distributions if you're producing an application for the mass market. Most customers prefer to use popular commercial or community distributions with standardized packaging processes. It reduces support headaches and often seems to make life easier. If you're interested in Gentoo Linux, don't forget to visit the project website at www.gentoo.org.

Roll Your Own

As you'll discover later in this book, it's possible to build your own Linux distribution from component parts. There are any number of reasons that you might want to do this — curiosity, the need for greater flexibility than is otherwise available, customizability, and so on. The fact is that many of the Embedded Linux devices on the market today are built entirely from scratch by the vendor producing the device. Needless to say, we do not encourage you to try building your own Linux distribution before you have become familiar with the internal packages, software, and utilities required by distributions in general.

The Linux From Scratch project is an example of one self-help guide you can use in building your own Linux distributions from scratch. Their website is www.linuxfromscratch.org. You can also check out automated distribution build tools such as PTXdist at <http://ptxdist.sf.net>.

Installing a Linux Distribution

Once you have decided upon which Linux distributions you will be working with, you'll need to set up at least one development machine. It's important to realize that you won't need to install every single

distribution you might want to later support when you start your development. Instead, choose one that you feel comfortable spending a lot of time working with as you try to get your software up and running. Later, you can port your software over to any other distributions that may be required. Don't forget that virtualization products – such as Xen and VMware – can greatly ease testing, as you can install any modern Linux distribution in its own virtualized sandbox away from your existing setup.

Later in this chapter, you'll find links to online groups and resources where you can discuss your choice of Linux distribution and ask any questions you may have while getting yourself set up.

Getting Hold of Linux

Most modern Linux distributions are supplied on CD or DVD media or are available to download in the form of CD or DVD images (ISOs) over the Internet. Distributions generally will use mirror sites to spread the load of the enormous numbers of people who wish to download CD or DVD images over their high-speed links. You can do your bit to help them out by always downloading from a mirror site that is geographically located near you. That way, you won't clog up international links unnecessarily with your large downloads — remember that Linux is international by its very nature.

Don't forget to check out BitTorrent as a means to harness peer-to-peer technology to speed up your download. Linux distributions covered under the terms of the GPL are freely redistributable, so many people have set up BitTorrent trackers to allow them to get faster downloads, while actually helping others speed up their downloads at the same time – look for explanations from vendor websites.

Be forewarned that downloading a particular distribution can take many hours, even on modern high-speed Internet connections. If you don't want to wait so long to download multiple CD or DVD images, you can often perform an online install instead. This process will take longer, but you will only install those packages that you select — so the installer won't need to retrieve as much data overall. To perform an online install, look for smaller network install CD images on vendor websites. These are often under 100MB in size and will download very quickly, while still allowing you to do a full install.

Of course, you might also elect to buy an off-the-shelf boxed product and save some of the time and hassle in downloading and burning media for yourself. If you choose to buy a copy of a commercial Linux distribution, look for a local Linux supplier that might be able to help you directly. They can come in handy later when you need to pick up any additional software — so use the opportunity to establish a relationship if you have the chance. You might also find that your local Linux user group additionally has a preferential deal with certain Linux vendors and suppliers for products for use by enthusiasts.

Determining Install-Time Package Selection

Installation of most modern Linux distributions is a smooth and painless process requiring that you answer just a few questions. Tell the installer the name you'd like to give to your Linux machine, what its network settings will be, and a few other details, and in no time at all, it'll be installing a lot of shiny software onto your machine. This is the ease with which a regular installation goes these days — certainly far removed from the days of having to build up the system from scratch yourself.

Most installers won't automatically include development tools when setting up a regular Linux desktop or server system. In particular, it is unusual to find the GNU toolchain and related build tools available out of the box with popular distributions — such as those from Red Hat, SuSE, or Ubuntu. You'll need to modify the package selection at install time to include what is usually labeled as “development tools” or similar. To do this, you might need to choose a custom install option, depending upon the specific version of the distribution that you are using. Check the documentation for advice.

Chapter 1: Working with Linux

Figure 1-1 shows the development packages being installed on a Fedora Core 5 system.



Figure 1-1

If you missed the opportunity to add development tools during system installation, you can go back and add in development tools at a later stage. This is usually best accomplished by using a graphical package management tool included with your distribution. Graphical package management tools such as yumex (Fedora), YaST (SuSE) and synaptic (Ubuntu) offer groups of related packages and ease the process of identifying what components you will need to install. If all else fails, you'll experience some strange errors when you try out some of the example code from this book — look out for missing tools.

Setting Up Your Development Environment

A newly installed Linux system will usually automatically load up a graphical desktop environment. Most Linux systems available today choose either of the GNOME or KDE graphical desktops (or in some cases both, allowing you to choose which you would like to use). Although this book attempts to be unbiased as possible, it is nevertheless not possible to cover all technologies to the same degree within a single volume. As a result, the authors have chosen to focus upon the GNOME desktop environment whenever it is necessary to talk specifically about desktop-related issues.

GNOME is the default graphical desktop environment used by both the Fedora and Ubuntu projects, but whatever your personal or organizational preference is, you should find the interfaces appear similar. You will quickly discover the administration and management tools located in the system menus, as well as those development tools that have been preinstalled by your distribution. Several distributions now ship with the Eclipse IDE development environment installed by default — a good place to start if you're familiar with other graphical development tools such as Microsoft Visual Studio on Windows.

Finding a Terminal

Linux systems, like other UNIX systems, are built upon many different tools and utilities that work together to get the job done. Although graphical desktop environments have become very popular over the last few years, it's still commonplace to perform everyday software source file editing and to drive software build processes entirely from within a system terminal. You can use a graphical development environment such as Eclipse, but it's a good idea to know how to work at the command line.

As you work through this book, most of the example code will include simple commands that you can use at the command line in order to build the software. You will usually find that a terminal is available to you via the system menus, or in some cases by right-clicking on your desktop and selecting Open Terminal from the menu. On Fedora systems, you'll need to install an extra system package (use the Software Updater tool in the System Tools menu, under the Applications menu) to have the terminal option readily available in your desktop menu — it's there by default on OpenSUSE and Ubuntu.

Editing Source Files

Throughout this book, you will find example source code that you can try out and modify for your own purposes. You'll find out more about how to build software on Linux systems in subsequent chapters. You'll also find many examples that are available from the website accompanying this book, which you can download in order to avoid typing them in each time. Despite this, you will clearly want to produce your own programs early on. It is, therefore, recommended that you find a text editor that you feel comfortable working with as you develop your Linux software.

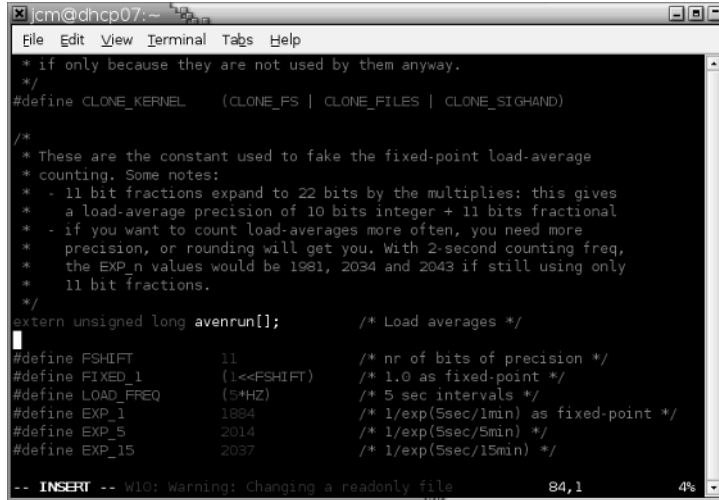
Most Linux developers choose to use popular editors such as vim (derived from the ancient UNIX vi editor) or GNU emacs (Richard Stallman's original GNU project editor). These work both from the command line and as graphical applications, depending upon the precise version you have installed. Each comes with a rich set of features that will enhance your productivity, as well as a set of documentation and tutorials to help you get up to speed quickly. For those who prefer a graphical editor, the GNOME and KDE desktops are supplied with several powerful alternatives.

It's worth noting the tradition of vi and emacs rivalry. Historically, vi and emacs users were mutually exclusive. Those who use one typically dislike the other with a passion (and other users of the other). There are few sources of contention more pointless than the editor flame wars started from time to time by people on mailing lists, but the sheer range of vi vs. emacs T-shirts and other merchandise available on the Internet should demonstrate the seriousness with which some people take these editor wars. It's never a good idea to try to understand precisely why people care so much about this — just live with it.

Whatever text editor you choose, don't try using a word processor such as Open Office writer or abi-word to edit program source code. While it is technically possible to do so, these tools usually mangle source and even when editing text files will attempt to embed various rich text formatting that will confuse the build tools you later use to build the software.

Chapter 1: Working with Linux

Figures 1-2 and 1-3 show examples of source files being edited with the vim and emacs text editors.



The screenshot shows the vim text editor in a window titled 'jcm@dhcp07:~'. The menu bar includes 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The editor displays a C source file with the following content:

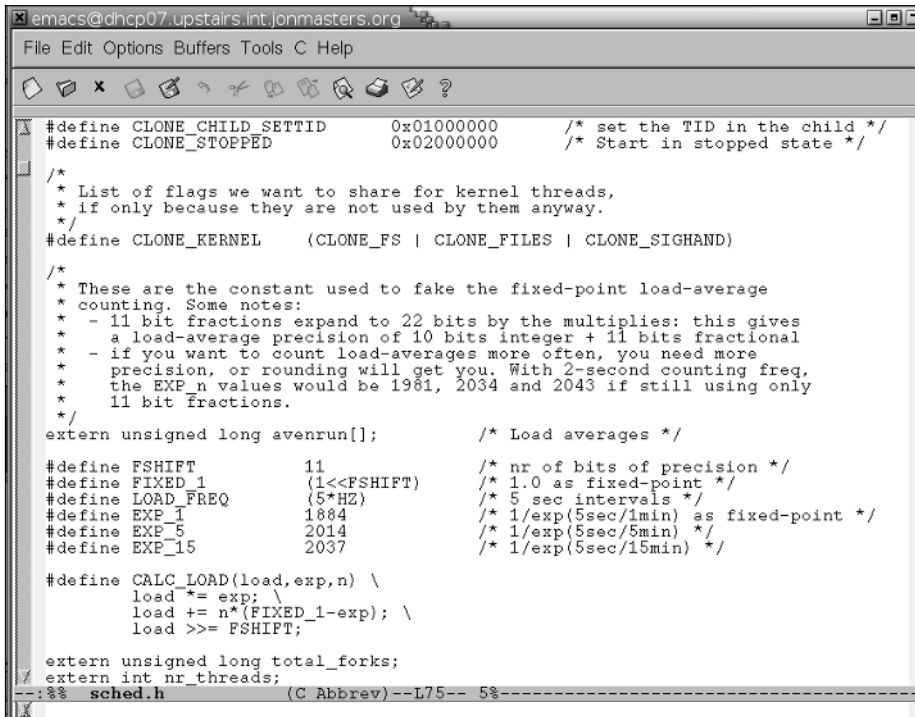
```
* if only because they are not used by them anyway.
*/
#define CLONE_KERNEL    (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)

/*
 * These are the constant used to fake the fixed-point load-average
 * counting. Some notes:
 * - 11 bit fractions expand to 22 bits by the multiplies: this gives
 *   a load-average precision of 10 bits integer + 11 bits fractional
 * - if you want to count load-averages more often, you need more
 *   precision, or rounding will get you. With 2-second counting freq,
 *   the EXP_n values would be 1981, 2034 and 2043 if still using only
 *   11 bit fractions.
 */
extern unsigned long avenrun[];          /* Load averages */

#define FSHIFT          11                /* nr of bits of precision */
#define FIXED_1         (1<<FSHIFT)     /* 1.0 as fixed-point */
#define LOAD_FREQ       (5*HZ)           /* 5 sec intervals */
#define EXP_1           1884             /* 1/exp(5sec/1min) as fixed-point */
#define EXP_5           2014             /* 1/exp(5sec/5min) */
#define EXP_15          2037             /* 1/exp(5sec/15min) */

-- INSERT -- W10: Warning: Changing a readonly file      B4,1      4%
```

Figure 1-2



The screenshot shows the emacs text editor in a window titled 'emacs@dhcp07.upstairs.int.jonmasters.org'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', and 'C Help'. The editor displays a C source file with the following content:

```
#define CLONE_CHILD_SETTID    0x01000000    /* set the TID in the child */
#define CLONE_STOPPED        0x02000000    /* Start in stopped state */

/*
 * List of flags we want to share for kernel threads,
 * if only because they are not used by them anyway.
 */
#define CLONE_KERNEL    (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)

/*
 * These are the constant used to fake the fixed-point load-average
 * counting. Some notes:
 * - 11 bit fractions expand to 22 bits by the multiplies: this gives
 *   a load-average precision of 10 bits integer + 11 bits fractional
 * - if you want to count load-averages more often, you need more
 *   precision, or rounding will get you. With 2-second counting freq,
 *   the EXP_n values would be 1981, 2034 and 2043 if still using only
 *   11 bit fractions.
 */
extern unsigned long avenrun[];          /* Load averages */

#define FSHIFT          11                /* nr of bits of precision */
#define FIXED_1         (1<<FSHIFT)     /* 1.0 as fixed-point */
#define LOAD_FREQ       (5*HZ)           /* 5 sec intervals */
#define EXP_1           1884             /* 1/exp(5sec/1min) as fixed-point */
#define EXP_5           2014             /* 1/exp(5sec/5min) */
#define EXP_15          2037             /* 1/exp(5sec/15min) */

#define CALC_LOAD(load,exp,n) \
    load *= exp; \
    load += n*(FIXED_1-exp); \
    load >>= FSHIFT;

extern unsigned long total_forks;
extern int nr_threads;
--:% sched.h (C Abbrev)--L75-- 5%
```

Figure 1-3

Using the Root Account

To avoid occasional accidental system damage — the removal of core system files, accidentally wiping out system utilities with your own, and so on — it's usual to do your everyday work as a regular user on your machine. A regular user has full access to his or her home directory (under `/home`) and can easily build and test out most regular application software. This is sufficient for most development tasks, but there are times when you will need to gain access to the administrative (root) account in order to modify global system settings, install test software, and generally to get the job done.

Rather than using your system entirely as the root user, or logging out and logging in as root whenever you need access to the root account, it's recommended that you use the `sudo` utility. `Sudo` enables you to run a single command as the root user, without running the risk of having to be logged in with such powers all of the time. It's amazing how easily you can accidentally trash a system with a single mistaken command as the root user. Hence, most developers generally use their own accounts.

To use `sudo`, you'll need to ensure that your regular user account is listed in `/etc/sudoers`. For example, the user account "jcm" can be granted `sudo` permission with the following entry:

```
jcm    ALL= (ALL)  ALL
```

This grants jcm permission to run any command as the root user (on any machine – there's only the local machine to worry about in most cases, but if you're on a network, check with your IS/IT folks). To actually run a command with root permissions, you can use the `sudo` command:

```
$ sudo whoami
root
```

You will be asked for a password, if you have not entered one recently.

The first time you use it, `sudo` warns you about the dangerous things you can do as a root user and then asks for a password, which may not be the same as your login password. On some distributions, `sudo` is configured to ask for the root account password by default, others will use your own login password in order to gain access to the `sudo` tool. You'll want to check your distribution's documentation or use the `UNIX man` command to find out more information about the local installation.

If you're working within a corporate environment, don't forget to notify your IT or IS department that you will require administrative access to your development machine. It'll save a lot of hassle later on, unless they specifically want to support you every time you need to use the root account.

Development Releases

Linux distributions usually have a development version that closely tracks ongoing development of the distribution itself. Such versions are updated far more often than their stable release counterparts. Stable distribution releases usually vary between 6 months and 18 months apart, while a development version might change on even a daily basis. The big three distributions — those from Red Hat, SuSE, and Ubuntu — all have unstable development releases available on a daily basis. You won't normally need to look at these, but it helps to know they're out there, so here's a quick overview of the alternatives.

Development releases of modern distributions are explained here for your interest and education. They may aid in some of your development decisions, but you should not directly build or develop production software on them. Changes occur frequently, making it extremely difficult to achieve reproducible results. Complete system breakage is also not that uncommon.

Chapter 1: Working with Linux

Red Hat calls their unstable Fedora release rawhide. It's available via the Fedora website, but you'll usually perform a "yum update" (after uncommenting the development YUM repository entries in `/etc/yum.repos.d`) from the most recent stable release in order to switch over, rather than trying to perform an install of rawhide directly. Rawhide is a hive of activity from which you can often garner some insight into what might make it into the next Fedora release — ultimately, that may even affect what goes into the Red Hat Enterprise product at some point in the future.

Novell calls its unstable OpenSUSE release Factory. It's available from the OpenSUSE website and can be installed using network bootable images that are available for download. You will need to follow the installation instructions carefully if you perform a network install as to do so necessitates changing various options early on in the boot process — before the YaST installer has even started. You can also upgrade using YUM (documented online), but that process is much newer as of this writing. Factory is updated on a semi-regular basis, ultimately feeding technology into SuSE Linux Enterprise Server.

Ubuntu, like Debian, is available in an unstable release. This release is updated frequently, whenever the packages within it are modified. Thus, it's sometimes the case that a given system is unable to perform an update to the latest unstable release at a particular moment in time. Unlike the other distributions mentioned here, Ubuntu and Debian provide an interim testing version of their distribution, which always contains packages that have been known to be usable after being released into unstable release of the distribution. You will usually "apt-get upgrade" your system to unstable after modifying `/etc/apt/sources.list`).

Scratch Boxes and Virtualization Technologies

As you become happier with Linux development and become more adventurous, you'll want to have a scratch box that you can just test out ideas on (or completely trash) without risking breaking anything it had installed. This is especially true if you later decide to write your own Linux kernel device drivers or otherwise modify critical system components — you don't want to be doing that on a machine you need to remain stable during that process. Many hackers resort to using old PCs for this purpose.

Virtualization technologies, such as VMware, qemu and Xen, can be useful ways to gain access to a large number of test virtual machines that you can happily trash all day long, all without actually having to buy any additional hardware. Not only is virtualization a good cost-saving idea, but it's also very practical when it comes to setting up standard test environments and sharing ideas with your colleagues. Most virtualization technologies will allow you to set up snapshots of systems configured in a particular way that you can then store or send to your coworkers via a network storage area of some kind.

VMware

VMware allows you to manage large collections of virtual machines using their proprietary graphical software. It's trivial to configure a new virtual machine and to then install a Linux distribution within it. Using VMware, you can easily install a range of different PC-based Linux distributions, all without actually changing the software on your machine. Great when you want to store preconfigured test machines or try out some experimental features that don't rely on having some specific custom hardware device, not so great for testing out your custom Linux kernel device drivers!

You can find out more information about VMware at www.vmware.com.

Qemu

Qemu is an open source virtualization technology that can be used to run Linux. It's entirely free, but somewhat more limited than proprietary offerings like VMware. Using the qemu command line utilities, you can create a virtual machine environment and then install a Linux distribution within it. Since qemu is covered by the GNU General Public License, it's possible to modify the software itself and add in interesting new capabilities. As you'll discover later in the book, possible modifications include custom virtualized hardware devices for which you can write your own device drivers — all without having to risk the stability of your regular development environment.

For more information about qemu, see the project website at www.qemu.org.

Xen

Over the past few years, there has been a growing interest in a virtualization technology known as Xen. At the time that this book is being written, Xen is making headlines most every other week. Like VMware, Xen is capable of running any operating system within a virtualized environment. Unlike VMware, Xen is also Free Software, and there are a variety of graphical configuration tools available for those who want to use them. Most recent Linux distributions include some level of specific support for building and configuring Xen virtualized environments that you can use to test out your software.

For more information about Xen, see the project website at www.cl.cam.ac.uk/Research/SRG/netos/xen.

Linux Community

One of the most important things to realize when developing software for Linux is that you are very much not alone. A large community of fellow developers exists all over the world. Many of these people are interested in trading stories or helping you out when you get yourself into a jam. There are numerous different ways in which you can contact fellow Linux users and developers — and you'll want to look into joining the wider community well before you run up against your first roadblock.

In addition to getting involved with the wider Linux community, you'll probably want to start reading one of the regular magazines. The oldest of these is the *Linux Journal* (www.linuxjournal.com), but at the time this book is being written, there are literally dozens of magazines from around the world.

Linux User Groups

Wherever you are in the world, you're probably located nearer to other Linux users than you may think. Most major towns or cities have their own Linux user group (LUG) that is made up of local Linux enthusiasts working in a wide range of different areas. Some members, like you, may be less familiar with Linux and be seeking advice from those who have been involved in the community for perhaps a decade or longer. In fact, some LUGs are well over 10 years old by this point.

You can find out more information about your local Linux user group via the Internet. Just type your local town or city name into Google's special Linux search engine at www.google.com/linux.

Mailing lists

Most of the Linux developers around the world today communicate new ideas, exchange software patches, and participate in general discussion via electronic mailing lists. There are so many different mailing lists on so many different topics that it would be impossible to cover them all in this book. Everything from the smallest subsystem of the Linux kernel to entire Linux distributions and even the most remote regions of the planet will have a mailing list of some kind. You are encouraged to join the mailing list from your local Linux user group and user lists provided by your distribution vendor as a starting point.

Throughout this book, you will find references to mailing lists and other similar resources that might help you to get more involved or better understand a particular topic.

IRC

Developers frequently wish to participate in more interactive discussion than a mailing list is designed to allow. IRC (Internet Relay Chat) facilitates the process of joining various channels on IRC networks around the world, and many groups that have a mailing list will also have an IRC channel of some kind to complement mailing list discussion. You can discover a wealth of online resources within IRC networks such as Freenode, OFTC, and others. Each of these is preconfigured into graphical IRC clients such as the xchat client that comes with most modern Linux distributions.

Private Communities

The Linux developer community isn't always quite as open as you may think it is. A number of closed groups do exist around the world in the name of facilitating discussion between bona fide core developers of a particular technology. These groups occasionally hold special events, but more often than not will simply communicate using nonpublic mailing lists and IRC networks. It's important to realize that Linux does have an open development process, despite the occasional private club.

An example of a private group is the many security groups around the world that look to quickly fix problems that are discovered in Linux software. They necessarily do not publish the precise details of what they're working on until they have found the security bug and made a coordinated release with any vendors and other third parties involved. This helps to reduce the number of security incidents. If you ever discover a security bug in Linux software, always use the appropriate channels to report it.

There is no Linux Cabal.

Key Differences

Linux isn't like other operating systems you may have encountered in the past. Most operating systems have been designed over a period of many years by a small team of highly skilled people. Those designers then handed over a specification document to software engineers for them to implement the design. Linux isn't about closed teams of any kind. Sure, there are many vendors working on Linux technology behind semi-closed doors, but the core of the development process happens out in the open for all to see — warts and all.

Having an open development process means that the whole world gets to dissect the implementation of various features within the Linux distributions and propose their own changes. Thus, Linux benefits from the “many eyeballs” scalability of allowing anyone to make a contribution — greatly exceeding the resources of even the largest proprietary software companies. Having an open development process means that it’s very hard to have bad ideas accepted into the core of your Linux system. Everyone makes mistakes, but the Linux community is usually very picky about what changes it will accept.

Linux Is Modular

A typical Linux system is built from many smaller components that work together to form the larger whole. Unlike Microsoft Windows, Linux explicitly breaks out even the smallest functionality into a separate dedicated utility. There are several different utilities designed solely to set the system clock, others to control the sound volume mixers on your sound card, still other dedicated tools for individual networking operations, and so on. Take a look in the standard `/bin` and `/usr/bin` directories on any Linux system, and you’ll see a few of these individual utilities for yourself.

Like many older UNIX systems, Linux is built upon a principal of KISS (“keep it simple, stupid”). This follows the principal that it’s better to do one thing and do it well, than try to overload functionality into giant monolithic programs. Unlike Microsoft Windows, Linux systems have been designed so that they are easily user modifiable. You are encouraged to customize your system in whatever manner you choose; such is the point of Free and Open Source software.

Throughout this book, you’ll see references to many tools that you may not have encountered previously. Don’t panic. As you’ll quickly discover, knowing the right people to turn to — the right resources and community groups — and using the documentation that is provided with every Linux system is often enough to get you out of a jam. Many so-called experts are in fact highly skilled Googlers who know how to get at the information that will help them to do whatever they need to.

Linux Is Portable

As you’ll discover in Chapter 3, Linux itself is one of the most portable operating systems available today. Linux distributions have been released for the smallest embedded devices like your cell phone, PDA, and digital video recorder (DVR) set-top box, while at the same time others support mainframe systems or supercomputers being used to process the human genome. Software written for Linux is often designed with portability in mind, and since it may automatically be built for a range of different target systems as part of being in a Linux distribution, it’s common to find portability issues early on.

When you write software for Linux, always consider whether your decisions will affect how portable your software will be in the future. Will you ever need to run it on 64-bit systems? Will you always have a full-featured graphical desktop environment based on GNOME, or might someone want to use your software on a resource-constrained embedded device? These are questions you should keep in mind so that you won’t have unnecessary surprises later on.

Linux Is Generic

The Linux kernel itself strives to be as generic as possible. That means that the same source code can be built to run on the smallest gadget or the largest mainframe with scalability built right in. There should not be any need to make fundamental adjustments in order to support this wide range of target systems,

Chapter 1: Working with Linux

because the software was built with this kind of flexibility in mind. Of course, certain features can and will be tuned for specific systems, but the core algorithms will remain the same. The same should be true of the majority of software released within modern Linux distributions.

Keeping things as generic as possible and allowing the users to decide how they will use a Linux system is one of the reasons that Linux has been so successful. You should always try to think about the ways that people might want to use your software and avoid incorporating unnecessary design constraints. You don't have to support people who want to use your software in nonrecommended ways, but don't artificially impose limits unless it's absolutely necessary.

Summary

In this chapter, you learned about working with Linux. You learned that the term Linux has different meanings in different contexts. Technically, Linux refers to the core operating system kernel written by Linus Torvalds and maintained by many thousands of skilled developers around the world. But Linux can also be used to refer to distributions of software that are built on top of Linus's original kernel. This software includes many thousands of tools and utilities, modern graphical desktop environments, and many other components that users expect to find in a complete modern OS.

The success of Linux over the past decade has a lot to do with the communities, which have helped to make it so popular as an alternative to big proprietary UNIX and other operating systems on the market today. You now know how to get involved in your local Linux User Group and how to make contact with a larger world of Linux developers, who are keen to help you on your way to becoming an expert. You also have gained an understanding of the many differences between Linux and other operating systems.